# GPGPUs in computational finance: Massive parallel computing for American style options

Gilles Pagès, Benedikt Wilbertz

# GPGPUs in computational finance: Massive parallel computing for American style options

Gilles Pagès* Benedikt Wilbertz†

January 17, 2011

**Abstract**

The pricing of American style and multiple exercise options is a very challenging problem in mathematical finance. One usually employs a Least-Square Monte Carlo approach (Longstaff-Schwartz method) for the evaluation of conditional expectations which arise in the Backward Dynamic Programming principle for such optimal stopping or stochastic control problems in a Markovian framework. Unfortunately, these Least-Square Monte Carlo approaches are rather slow and allow, due to the dependency structure in the Backward Dynamic Programming principle, no parallel implementation; whether on the Monte Carlo level nor on the time layer level of this problem.

We therefore present in this paper a quantization method for the computation of the conditional expectations, that allows a straightforward parallelization on the Monte Carlo level. Moreover, we are able to develop for AR(1)-processes a further parallelization in the time domain, which makes use of faster memory structures and therefore maximizes parallel execution.

Finally, we present numerical results for a CUDA implementation of this methods. It will turn out that such an implementation leads to an impressive speed-up compared to a serial CPU implementation.

*Keywords: Voronoi Quantization, Markov chain approximation, CUDA, Parallel computing for financial models, Stochastic control.*

## 1 Introduction

The pricing of American style and multiple exercise options consists of solving the optimal stopping problem

$$V = \operatorname{esssup}\Big\{\mathbb{E}\big(\varphi_\tau(X_\tau)\big|\mathcal{F}_0\big) : \tau \text{ is a } (\mathcal{F}_k)\text{-stopping time}\Big\}$$

*Laboratoire de Probabilités et Modèles aléatoires, UMR 7599, Université Paris 6, case 188, 4, pl. Jussieu, F-75252 Paris Cedex 05. E-mail: gilles.pages@upmc.fr

†Laboratoire de Probabilités et Modèles aléatoires, UMR 7599, Université Paris 6, case 188, 4, pl. Jussieu, F-75252 Paris Cedex 05. E-mail: benedikt.wilbertz@upmc.fr

for an adapted stochastic process $(X_k)_{0\leq k\leq n}$ on a filtered probability space $(\Omega, (\mathcal{F}_k)_{0\leq k\leq n}, \mathbb{P})$ and obstacle functionals $\varphi_k, 0 \leq k \leq n$.

It is well known (see e.g. [14]) that $V$ is given by the solution $V_0$ to the *Backward Dynamic Programming (BDP) Principle*

$$\begin{aligned} V_n &= \varphi_{t_n}(X_n) \\ V_k &= \max\Big(\varphi_{t_k}(X_k);\, \mathbb{E}\big(V_{k+1}\big|\mathcal{F}_k\big)\Big),\ 0 \leq k \leq n-1. \end{aligned} \tag{1}$$

We focus here on the case of an adapted Markov chain $(X_k)$, so that it holds $\mathbb{E}(V_{k+1}|\mathcal{F}_k) = \mathbb{E}(V_{k+1}|X_k)$. Then the main difficulty of solving (1) by means of Monte Carlo methods lies in the approximation of the conditional expectations $\mathbb{E}(V_{k+1}|X_k)$. This is usually accomplished by a Least Squares regression as proposed by the Longstaff-Schwartz method. Following [6, 11] and [15] the main steps of this procedure consists of

- Simulating $M$ paths of $(X_k)$ (forward step)
- Starting at $k = n-1$, approximate $f_k(x) = \mathbb{E}(V_{k+1}|X_k = x)$ by a Least Squares regression and proceed backwards to 0. (backward step)

From a practical point of view, the most expensive tasks are clearly the repeated Least Square regressions on the huge number of Monte Carlo paths. Due to the sequential dependency structure of the Backward Dynamic Programming formula, the collection of the Least Squares problems as a whole cannot be solved in parallel, but has to be processed in strict sequence. Moreover, it is not an easy task to solve the single Least Square problems efficiently in parallel.

We therefore present in this paper a Quantization Tree algorithm, which handles the most part of the work in a forward step which can be easily parallelized on the Monte Carlo level (pathwise) as well as on the time layer level. Therefore, this approach is well suited for the use of massive parallel computing devices like GPGPUs. Using this approach, the subsequent backward processing of the BDP principle becomes straightforward and negligible in terms of computational costs when compared to the Least Squares backward step.

## 2 The Quantization Tree Algorithm

The Quantization Tree algorithm is an efficient tool to establish a pathwise discretization of a discrete-time Markov chain (see e.g. [1, 2, 3] or [5]). Such a discretization can be used to solve optimal stopping or control problems, as they occur in the evaluation of financial derivatives with non-vanilla exercise rights. In this paper, we focus on a fast computation of the transition probabilities in a Quantization Tree by means of `GPGPU`-devices, which make this approach suitable for time-critical online computations.

Therefore, let $(X_k)_{0\leq k\leq n}$ be a discrete-time $L^2$-Markov chain on a filtered probability space $(\Omega, (\mathcal{F}_k)_{0\leq k\leq n}, \mathbb{P})$ with values in the vector space $(\mathbb{R}^d, \mathcal{B}^d)$. This vector space shall be endowed with an appropriated norm (often Euclidean

norm). For each time-step $k$ we furthermore assume to a have a quantization grid

$$\Gamma_k = (x_1^k, \dots, x_{N_k}^k)$$

of size $N_k$.

This means that $\Gamma_k$ provides a discretization of the state space of the r.v. $X_k$, which is supposed to minimize the quadratic quantization error

$$\mathbb{E} \min_{1 \le i \le N_k} \|X_k - x_i^k\|^2 \tag{2}$$

over all possible grids $\Gamma_k \subset \mathbb{R}^d$ with size $|\Gamma_k| \le N_k$. (See [8] for a comprehensive introduction to quantization of probability distributions.)

For a grid $\Gamma_k$, let $\big(C_i(\Gamma_k)\big)_{1 \le i \le N_k}$ be a *Voronoi Partition* of $\mathbb{R}^d$ induced by the points in $\Gamma_k$, i.e.

$$C_i(\Gamma_k) \subset \big\{ y \in \mathbb{R}^d : \|y - x_i^k\| \le \min_{1 \le j \le N_k} \|y - x_j^k\| \big\}.$$

We then call the mapping

$$z \mapsto \sum_{i=1}^{N_k} x_i \mathbf{1}_{C_i(\Gamma_k)}(z)$$

the *Nearest Neighbor projection* of $z$ onto $\Gamma_k$.

This Nearest Neighbor projection defines in a natural way the *Voronoi Quantization*

$$\widehat{X}_k^{\Gamma_k} = \sum_{i=1}^{N_k} x_i \mathbf{1}_{C_i(\Gamma_k)}(X_k),$$

which obviously provides a discrete r.v. with not more than $N_k$ states and

$$\mathbb{E}\|X_k - \widehat{X}_k^{\Gamma_k}\|^2 = \mathbb{E} \min_{1 \le i \le N_k} \|X_k - x_i^k\|^2.$$

Defining the cartesian product quantizer

$$\Gamma = \prod_{k=0}^{n} \Gamma_k$$

we arrive at a path discretization of the Markov chain $(X_k)$ with $|\Gamma| \le \prod_{k=0}^n N_k$ paths, which we will call the *Quantization Tree* (see Figure 1).

To equip $\Gamma$ with a probability distribution, we introduce the transition probabilities

$$\begin{aligned} \pi_{ij}^k &= \mathbb{P}\big(\widehat{X}_k^{\Gamma_k} = x_j^k \,|\, \widehat{X}_{k-1}^{\Gamma_{k-1}} = x_i^{k-1}\big) \\ &= \mathbb{P}\big(X_k \in C_j(\Gamma_k) \,|\, X_{k-1} \in C_i(\Gamma_{k-1})\big). \end{aligned} \tag{3}$$

If the marginal distributions of $(X_k)$ are Gaussian and the norm is the canonical Euclidean norm, grids which minimize (2) are precomputed and available

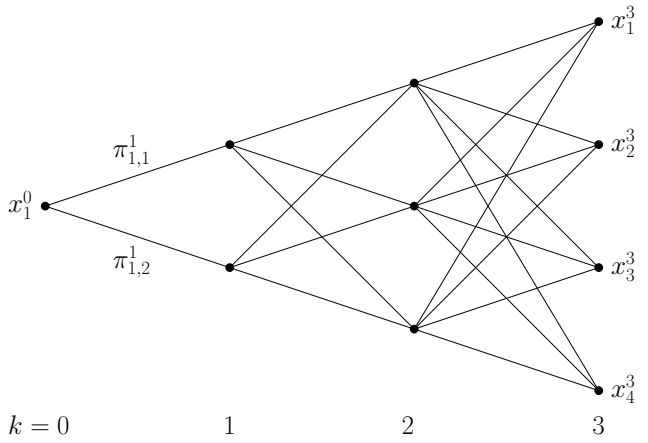


Figure 1: A Quantization Tree $\Gamma$

at [13]. Otherwise, some sub-optimal grids, matching the first two moments of $X_k$, can be employed at the price of not achieving the full optimal convergence rate.

Nevertheless, the true difficulties of this approach actually consist in the computations of the transition probabilities $\pi_{ij}^k$. These probabilities are usually so strongly connected to the individual choice of the Markov chain $(X_k)$ that they cannot be precomputed like the above quantization grids or approximated by simple means.

We therefore have to perform a Monte-Carlo (MC) simulation of the Markov chain $(X_k)$ in order to estimate the transition probabilities $\pi_{ij}^k$. Since these MC simulations can be quite time consuming, we will take advantage of the massive parallel computing capabilities of nowadays `GPGPU`-devices and reduce the computational time for the estimation of the transition probabilities to a level that actually is acceptable for time-critical applications in financial practice.

As the Quantization Tree $\Gamma$ exhibits a pathwise approximation of the Markov chain $(X_k)$, we may numerically solve on $\Gamma$ stochastic control or optimal stopping problems like they occur e.g. in the valuation of options with non-vanilla right exercises.

In [2], the optimal stopping problem

$$V = \operatorname{esssup}\Big\{\mathbb{E}\big(\varphi_\tau(X_\tau)\big|\mathcal{F}_0\big) : \tau \text{ a } (\mathcal{F}_k)\text{-stopping time}\Big\} \tag{4}$$

with a payoff function $\varphi_t(x) = \big(s_0 \exp\big((r - \sigma^2/2)t + \sigma x\big) - K\big)^+$ and $(X_k)$ a $d$-dimensional time-discretized Brownian motion is solved to approximate American option prices.

In [3], the authors employ the Quantization Tree to solve the stochastic

control problem

$$P(Q) = \operatorname{esssup}\Big\{ \mathbb{E}\Big( \sum_{k=0}^{n-1} q_k v_k(X_k) \Big| \mathcal{F}_0 \Big) : \forall k = 0, \dots, n-1 : \\ q_k \colon (\Omega, \mathcal{F}_k) \to [0,1],\ \sum_{k=0}^{n-1} q_k \in [Q_{\min}, Q_{\max}] \Big\}, \tag{5}$$

where $v_k$ can be interpreted as a payoff function and the couple $Q = (Q_{\min}, Q_{\max})$ provides some global constraints on the cumulated consumption $\sum_{k=0}^{n-1} q_k$, so that (5) yields the fair value of a swing option, which is an important derivative in energy trading.

Concerning the Quantization Tree algorithm, note that $\Gamma$ contains such a huge number of paths (e.g. at least $100^{365}$ in the example below) that it is impossible to process above problems in a path-wise manner.

Therefore, one usually resorts on the *Backward Dynamic Programming (BDP) Principle*, which allows a time-layer wise proceeding. This approach yields a complexity of $C \sum_{k=1}^{n} N_{k-1} N_k$, i.e. increases only linearly in $n$.

In case of the optimal stopping problem (4), the true BDP-principle can be approximated by setting

$$\begin{aligned} \widehat{V}_n &= \varphi_{t_n}(\widehat{X}_n^{\Gamma_n}) \\ \widehat{V}_k &= \max\Big( \varphi_{t_k}(\widehat{X}_k^{\Gamma_k}); \mathbb{E}\big(\widehat{V}_{k+1} \big| \widehat{X}_k^{\Gamma_k}\big) \Big),\ 0 \le k \le n-1, \end{aligned}$$

so that the $\mathcal{F}_0$ measurable r.v. $\widehat{V}_0$ yields an approximation for $V$. Doing so we somehow "force" the Markov property of the Quantization sequence $(\widehat{X}_k^{\Gamma_k})$.

In case of the stochastic control problem (5), it was shown in [4] that there exists a bang-bang control for (5), so that the BDP-principle leads to

$$\begin{aligned} \widehat{P}_n &\equiv 0 \\ \widehat{P}_k(Q^k) &= \max\Big\{ x v_k(\widehat{X}_k^{\Gamma_k}) \\ &\qquad + \mathbb{E}\big(\widehat{P}_{k+1}(\chi^{n-k-1}(Q^k, x)) \big| \widehat{X}_k^{\Gamma_k}\big), x \in \{0,1\} \cap I_{Q^k}^{n-k-1} \Big\}, \end{aligned}$$

where the set $I_Q^k$ and the function $\chi^k(Q,x)$ ensure to keep consumption within the global constraints $[Q_{\min}, Q_{\max}]$.

In both cases, we have to evaluate conditional expectations $\mathbb{E}\big(f(\hat{X}_{k+1}) | \hat{X}_k\big)$, which reduce on $\Gamma$ to

$$\mathbb{E}\big(f(\widehat{X}_{k+1}^{\Gamma_{k+1}}) \big| \widehat{X}_k^{\Gamma_k} = x_i^k\big) = \sum_{j=1}^{N_{k+1}} f(x_j^{k+1})\, \pi_{ij}^k.$$

Concerning the approximation error for this approach, assume that the $v_k$ are Lipschitz-continuous and that $(X_k)$ has Lipschitz-Feller transition kernels.

We then get in case of a trivial $\sigma$-field $\mathcal{F}_0$ for a constant $C > 0$ (see [4], Thm 3)

$$|P(Q) - \widehat{P}_0(Q)| \leq C \sum_{k=0}^{n-1} \Big( \mathbb{E}\|X_k - \widehat{X}_k^{\Gamma_k}\|^2 \Big)^{1/2}.$$

# 3 Swing options in the Gaussian 2-factor model

We will now focus on the implementation of the Quantization Tree algorithm for the valuation of Swing options in a Gaussian 2-factor model and present in detail the computation of the transition probabilities using `CUDA` on a `GPGPU`-device.

In this model, the dynamics of the underlying are given as

$$S_t = s_0 \exp\left( \sigma_1 \int_0^t e^{-\alpha_1(t-s)} dW_s^1 + \sigma_2 \int_0^t e^{-\alpha_2(t-s)} dW_s^2 - \frac{1}{2}\mu_t \right)$$

for Brownian Motions $W^1$ and $W^2$ with some correlation parameter $\rho$.

Having introduced the time discretization $t_k = k/n$, $k = 0, \ldots, n$, we consider the 2-dimensional Ornstein-Uhlenbeck process

$$X_k = \Big( \int_0^{t_k} e^{-\alpha_1(t_k - s)} dW_s^1, \int_0^{t_k} e^{-\alpha_2(t_k - s)} dW_s^2 \Big). \tag{6}$$

This Markov chain admits a useful representation as a *first-order autoregressive (AR-1)*-process:

**Proposition 1** *For $(X_k)$ from (6) it holds*

$$X_{k+1} = A_k X_k + T_k \epsilon_k, \qquad k = 0, \ldots, n-1,$$

*where $A_k$ and $T_k$ are deterministic matrices and $(\epsilon_k)$ is an i.i.d. standard normal sequence.*

In order to estimate the transition probabilities

$$\begin{aligned} \pi_{ij}^k &= \mathbb{P}\big( X_k \in C_j(\Gamma_k) \,|\, X_{k-1} \in C_i(\Gamma_{k-1}) \big) \\ &= \frac{\mathbb{P}\big( X_k \in C_j(\Gamma_k) \cap X_{k-1} \in C_i(\Gamma_{k-1}) \big)}{\mathbb{P}\big( X_{k-1} \in C_i(\Gamma_{k-1}) \big)}, \end{aligned}$$

we will therefore simulate $M$ samples of $(X_k)$ according to Proposition 1 and perform in each time-layer $k$ a Nearest Neighbor search to identify the Voronoi cell $C_j(\Gamma_k)$ in which $X_k$ falls.

Using the additional counters $p_{ij}^k$ and $p_i^k$, a serial implementation for the estimation of $\pi_{ij}^k$ is given by Algorithm I.

We will adopt a numerical scenario, which has already proven in [5] to produce accurate results for the valuation of Swing options. Thus we set

**# MC-Samples:** $M = 100.000$

**Algorithm I**

**for** $m = 1, \ldots, M$ **do**
  # Initialization
  $x \leftarrow x_0$, $i \leftarrow 0$, $p_1^i \leftarrow 1$
  **for** $k = 1, \ldots, n$ **do**
    Simulate $\epsilon_k$
    $x \leftarrow A_k x + T_k \epsilon_k$
    Find NN-Index $j$ of $x$ in $\Gamma_k$
    Set
      $p_{ij}^k$ += 1
      $p_j^{k+1}$+= 1
    $i \leftarrow j$
  **end for**
**end for**
Set $\pi_{ij}^k \leftarrow \frac{p_{ij}^k}{p_i^k}, \quad 1 \le i, j \le N_k, 1 \le k \le n.$

**# Exercise days:** $n = 365$

**Grid size:** $N = N_k = 100 - 500$ for $k = 1, \ldots, n$.

This setting results in a computational time of $30-90$ seconds for non-parallel estimation of the transition probabilities on a `Intel Core i7` CPU@2.8GHz and $N = 100$ to 500.

Since any parallel implementation of the above algorithm has to perform actually the following steps

1.) generation of the independent random numbers $\epsilon_k$

2.) a Nearest Neighbor search

3.) updating the counters $p_{ij}^k, p_i^k$,

we will discuss these tasks in more detail with respect to an implementation for `CUDA`.

The amount of data which has to be processed in these steps when using single precision floating-point numbers is summarized in Table 1.

Table 1: Amount of data to be processed for $N = 100 - 500$.

| | per layer $k$ | total |
|---|---|---|
| # Random numbers | 100k | 36.5M |
| # Nearest Neighbor searches | 100k | 36.5M |
| size of $\pi_{ij}^k$ and $p_{ij}^k$ | 40kB - 1MB | 15 - 365MB |
| size of grids $\Gamma_k$ | 800Byte - 4kB | 285kB - 1.5MB |

## 3.1 Random number generation

The challenge of random number generation on parallel devices consists in modifying the sequential random number generator algorithm in such a way, that the original sequence $\{x_n, n = 1, \ldots, M\}$ with $M = k \cdot s$

- is generated in independent blocks of size $s$, i.e. $k$ streams $\{x_{n\cdot s+i},\, i = 1, \ldots, s\}$, where $n = 0, \ldots, k-1$ (block approach)

or

- can be partitioned through a skip-ahead procedure, i.e. one generates independently $s$ streams $\{x_{n+i\cdot s},\, i = 0, \ldots, k-1\}$ for $n = 1, \ldots, s$ (skip-ahead)

The block-approach can be accomplished by generating a well chosen sequence of seed values to start the parallel computation of the random number streams. In contrast to this, for the skip-ahead approach we have to modify the main iteration of the random number generator itself . Nevertheless, this modification can be easily carried out for linear congruential random number generators

$$x_{n+1} \equiv a x_n + c \mod 2^m,$$

For this kind of generator it holds

$$x_{n+s} \equiv A x_n + C \mod 2^m$$

with $A = a^s$ and $C = \sum_{i=0}^{s} a^i c$. Thus, once the coefficients $A$ and $C$ are computed, the generation of the subsequence $\{x_{n+is}, i \in \mathbb{N}\}$ is as straightforward as it is for $\{x_n, n \in \mathbb{N}\}$.

As a first parallel random number generator, we have implemented a parallel version of `drand48` in `CUDA`, which operates in 48bit arithmetic.

A slightly more sophisticated variant of this random number generator is given by L'Ecuyer's Multiple Recursive Generator MRG32k3a (cf. [9])

$$\begin{aligned} x_n^1 &= (1403580\, x_{n-2}^1 - 810728\, x_{n-3}^1) \mod m_1 \\ x_n^2 &= (527612\, x_{n-1}^2 - 1370589\, x_{n-3}^2) \mod m_2 \\ x_n &= (x_n^1 - x_n^2) \mod m_1 \end{aligned}$$

for $m_1 = 2^{32} - 209$ and $m_2 = 2^{32} - 22853$.

Here, it is again possible to precompute constants (matrices) to generate the skip-ahead sequence $\{x_{n+is}, i \in \mathbb{N}\}$ efficiently (see [10]). An implementation in `CUDA` of this method is given by the GPU-Library of NAG.

A third kind of random number generators for `CUDA` is given by Marsaglia's XORWOW generator in the `CURAND`-Library of `Cuda Toolkit 3.2`. As described in [12] one easily may compute starting seed values for a block approach and the random numbers sequence is then given by very small number of fast bit-shifts and XOR-operations. To be more precise the initialization procedure of the `CURAND`-Library computes starting values for the blocks which correspond to $2^{67}$ iterations of the random engine. Moreover the main iteration of the random number generator for the state variables `v,w,x,y,z` reads

```
unsigned int curand()
{
    unsigned int t;
    t = ( x ^ (x >> 2) );
    x = y;
    y = z;
    z = w;
    w = v;
    v = (v ^ (v << 4))^(t ^ (t << 1));
    d += 362437;
    return v + d;
}
```

To illustrate the performance of these three random number generators we have chosen a Monte Carlo simulation with a very simple integrand to illustrate the performance in simulations where the function evaluation is very cheap. To be more precise, we estimated $\pi = 3.14159265...$ by a Monte Carlo simulation for $\frac{1}{2}\lambda^2\big(B_{l^2}(0,1)\big)$ using $M = 10^9$ random numbers.

The results for a `NVIDIA GTX 480` device and `CUDA 3.2` are given in Table 2. The mean and the standard deviation of the MC-Estimator were computed from a sample of size 500.

| RNG engine | computational time | mean | std. Dev. |
|---|---|---|---|
| drand48 | 0.2562 sec | 3.141590 | 5.2585e-05 |
| MRG32k3a | 0.2573 sec | 3.141594 | 5.20932e-05 |
| CURAND | 0.2085 sec | 3.141592 | 5.03272e-05 |

Table 2: Results for a Monte Carlo estimation of $\pi = 3.14159265...$

One recognizes that the XORWOW generator from the `CURAND`-library slightly outperforms the two linear congruential implementations, since the XORWOW-step can be processed more efficiently than a modulo operation. Nevertheless the differences between all three random number generator are rather marginal.

Especially, when we have in mind, that the original problem of swing option pricing needs only 35M random numbers in total, the generation of this amount of random numbers becomes negligible compared to the time spent for the nearest neighbor searches.

## 3.2 Nearest Neighbor search

For each MC-realization $X_k$ we have to perform a Nearest Neighbor search in every time-layer $k$ to determine the Voronoi cell $C_j(\Gamma_k)$ in which $X_k$ falls.

These Nearest Neighbor searches can be performed completely independent of each other, so we implemented them as sequential procedures and only have to pay attention to a proper adaption to the `CUDA`-compute capabilities.

Note here that we cannot employ the CUDA built-in texture fetch methods for this task, since the grids $\Gamma_k$ do in general not consist of a lattice of integer numbers.

From an asymptotical point of view, the *kd*-tree methods (cf [7]) obtain the fastest results for Nearest Neighbor searches of $O(\log N)$-time. Unfortunately, all these divide & conquer-type approaches heavily rely on recursive function calls; a programming principle which was introduced only very recently in the CUDA Compute Capability 2.x specification. Alternatively, one may implement a simple brute force Nearest Neighbor search of $O(N)$-time complexity.

The results for 36.5M NN Searches of a random number in a 2-dimensional grid can be found in Table 3. It is striking that the brute force approach

| $N$ | brute force | *kd*-tree |
|---|---|---|
| 100 | 0.09 sec | 3.56 sec |
| 250 | 0.23 sec | 5.14 sec |
| 500 | 0.41 sec | 6.59 sec |

Table 3: Computational time for 36.5M Nearest neighbor searches on a NVIDIA GTX 480 device

outperforms the *kd*-tree method in this setting by a huge factor, even though it suffers from a sub-optimal asymptotic behavior.

Further analysis revealed that, when using the same random number for the search in all threads of a given block, the *kd*-tree approach took in the same setting only 0.25 to 0.34 sec ($N = 100$ to 500). The dramatic slowdown of Table 3, where the NN Search is performed for different random numbers in each single thread, must be caused by a very inhomogeneous branching behavior of the single threads during the *kd*-tree traversal, which prevents the GPGPU-scheduler of distributing the threads efficiently.

We will therefore use in the sequel the brute force approach for the further numerical experiments.

## 3.3 Updating $p_{ij}^k$

As soon as we have determined the Voronoi cells $C_i(\Gamma_{k-1})$ and $C_j(\Gamma_k)$ in which a realization of $(X_{k-1}, X_k)$ falls, we have to increase the counter $p_{ij}^k$.

Since, in a parallel execution of steps 3.1. and 3.2., it can happen that two threads try to update the same counter $p_{ij}^k$ at the same time, we arrive at the classical situation of a race condition.

Consequently, such a situation would lead to an undetermined result for the counter $p_{ij}^k$, which practically means that we randomly lose parts of the Nearest Neighbor search results.

To avoid this race condition, we are forced to employ memory locks, which are implemented in CUDA by means of atomic operations. Hence, we have to increment $p_{ij}^k$ by calling the CUDA-function

```
int     atomicAdd(int* address, int val);.
```

The resulting parallel procedure is stated as Algorithm II.

**Algorithm II**

**for** $m = 1, \ldots, M$ **do in parallel**
  # Initialization
  $x \leftarrow x_0$, $i \leftarrow 0$, $p_1^i \leftarrow 1$
  **for** $k = 1, \ldots, n$ **do**
    Simulate $\epsilon_k$
    $x \leftarrow A_k x + T_k \epsilon_k$
    Find NN-Index $j$ of $x$ in $\Gamma_k$

    atomic increment $p_{ij}^k$
    atomic increment $p_j^{k+1}$
    $i \leftarrow j$
  **end for**
**end for in parallel**
Synchronize threads
Set in parallel $\pi_{ij}^k \leftarrow \frac{p_{ij}^k}{p_i^k}, \quad 1 \le i, j \le N_k, 1 \le k \le n.$

# 4 Numerical results

One of the key points in an efficient `CUDA`-implementation is the choice of the proper memory structure for the individual data. Table 4 lists the available memory types in `CUDA` Compute Capability 1.x.

| | | |
|---|---|---|
| local memory | **not** cached | 16kB per thread |
| constant memory | cached | 64kB per device |
| shared memory | n/a | 16kB per block |
| global memory | **not** cached | $\approx$ 1GB per device |

Table 4: Memory types for `CUDA` compute capability 1.x

Note that *shared memory* is (beneath the processor registers) the fastest memory available in `CUDA`, since it resides very close to the processor cores. There are 16kB of shared memory available per Multiprocessor, whose content is read- and writable by any thread in the same block of a grid.

The other memory types in Table 4 are about 400 times slower than shared memory except *constant memory* which is cached and therefore achieves a similar read performance as shared memory.

Taking into account the sizes of the arrays $\pi_{ij}^k, p_{ij}^k$ and $\Gamma_k$ from Table 1, there is no other possibility for the above algorithm than to place all the arrays in global memory, since any thread has to access the arrays $\pi_{ij}^k, p_{ij}^k$ and $\Gamma_k$ for any $k$, $1 \le k \le n$.

The fact that these arrays have to reside in global memory especially slows down the Nearest Neighbor searches, which rely on a fast access to the grid points of $\Gamma_k$.

We therefore present another approach, which maximizes the parallel execution by splitting up the problem into smaller parts, that can make use of faster memory.

Note that due to Proposition 1 we can directly simulate the couple $(X_k, \varepsilon_k)$ in order to get a realization of $(X_k, X_{k+1})$ without the need of generating $X_l$, $l < k$.

Thus, if we accept to generate twice the amount of random numbers and double the number of Nearest Neighbor searches, we arrive at Algorithm III.

**Algorithm III**

**for** $k = 1, \ldots, n$ **do in parallel**
  **for** $m = 1, \ldots, M$ **do in parallel**
    Simulate $X_k, \epsilon_k$

    Find NN-Index $i$ of $X_k$ in $\Gamma_k$
    Find NN-Index $j$ of $A_k X_k + T_k \epsilon_k$ in $\Gamma_{k+1}$

    atomic increment $p_{ij}^k$
    atomic increment $p_i^k$
  **end for in parallel**
  Synchronize
  Set in parallel $\pi_{ij}^k \leftarrow \frac{p_{ij}^k}{p_i^k}, \quad 1 \leq i, j \leq N_k$
**end for in parallel**

Here, we do not only parallelize with respect to the MC-samples (pathwise), but also with respect to the time-layer $k$. Therefore, we are able to perform the whole MC-simulation of a given time-layer $k$ (i.e. the inner loop) on a single Multiprocessor (i.e. within a single block in `CUDA`-terminology).

Hence, we can store the involved grids $\Gamma_k$ and $\Gamma_{k+1}$ entirely in shared memory and benefit from a huge performance gain.

This can be seen in Table 5 and Figure 2, which demonstrates that the shared memory implementation - performing even twice as many Nearest Neighbor searches - is still significantly faster than the usual pathwise parallelization for `CUDA` Compute Capability 1.x.

| $N$ | 100 | 250 | 500 |
|---|---|---|---|
| Algorithm II | 0.82 sec | 1.25 sec | 1.83 sec |
| Algorithm III | 0.31 sec | 0.68 sec | 1.38 sec |

Table 5: Computational times for the transition probabilities on a `NVIDIA GTX 295` device

All the computations for `CUDA` Compute Capability 1.x were performed on a `NVIDIA GTX 295` GPGPU, `CUDA` Toolkit 2.3 and NVIDIA X-Driver 190.53

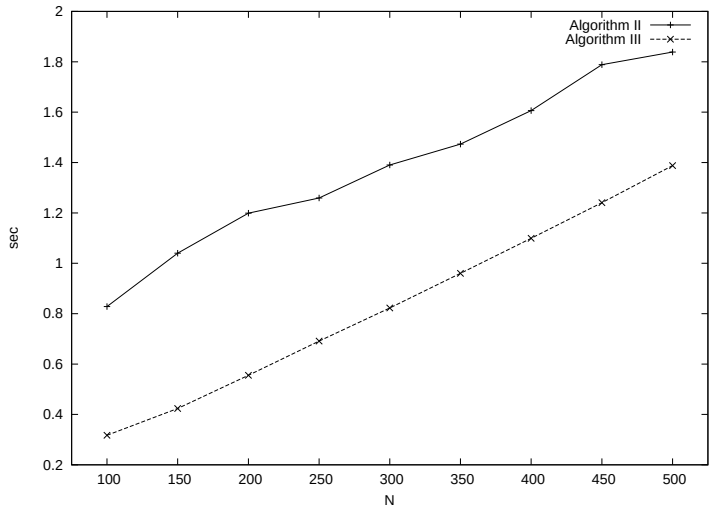


Figure 2: Linear performance of Algorithms II & III with respect to $N$ on a `NVIDIA GTX 295` device

for 64bit Linux. The running times in Table 5 also include the transfer of the transition probabilities $\pi_{ij}^k$ back to the host CPU.

Furthermore, we have chosen in all examples 256 - 512 threads per block and overall 365 - 400 blocks. This choice was optimal for our setting. Note here, that the shared memory algorithm performs $73 \cdot 10^6$ Nearest Neighbor searches in a 2-dimensional grid. Assuming that the brute force Nearest Neighbor search

$$\left(\min < (x_1 - y_1)^2 + (x_2 - y_2)^2\right)$$

for each grid point is equivalent to 6 FP-operations (3 additions, 2 multiplications, 1 comparison), we already arrive for $N = 500$ at a computing power of approx. 175 GFLOPS only for the Nearest Neighbor searches (the pure kernel execution takes in this case 1.25sec). Compared to the peak performance of 895 GFLOPS for one unit in the `NVIDIA GTX 295`-device, this fact underlines that our implementation exploits a great amount of the theoretically available computing power of a `GPGPU`-devices.

## 4.1 Progress in hardware: the Fermi-architecture

With the arrival of `CUDA` Compute Capability 2.x and the Fermi-architecture, there are now L1- and L2 caches available of up to 48kB per block. It turned out that this change in hardware design has strong implications on the performance of Algorithm II. As it can be seen in Table 6 and Figure 3, the new cache can nearly completely compensate the advantage of the shared memory usage in Algorithm III. Moreover, both parallelizations differ roughly by a factor of two which is caused by the fact that algorithm III has to perform twice the

| $N$ | 100 | 250 | 500 |
|---|---|---|---|
| Algorithm II | 0.11 sec | 0.30 sec | 0.63 sec |
| Algorithm III | 0.21 sec | 0.50 sec | 0.99 sec |

Table 6: Computational times for the transition probabilities on a `NVIDIA GTX 480` device

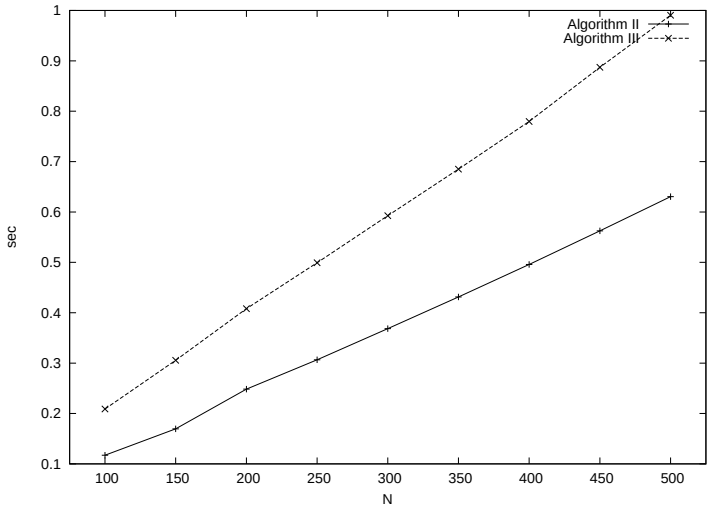


Figure 3: Performance of Algorithms II & III with respect to $N$ on a `NVIDIA GTX 480` device

number of Nearest Neighbor searches than Algorithm II. The computations for `CUDA` Compute Capability 2.x were performed on a `NVIDIA GTX 480` GPGPU, `CUDA` Toolkit 3.2 and NVIDIA X-Driver 260.19.29 for 64bit Linux

# 5 Conclusion

We have shown in this paper that the use of `GPGPU`-devices is quite efficient for the estimation of transition probabilities in a Quantization Tree. Although we resorted for the Nearest Neighbor search, which is the most compute intensive part of the algorithm, to the sub-optimal brute-force approach, we could achieve by means of the massive computing power of a `GPGPU`-device a speed-up of factor 200 compared to a serial CPU implementation. Those implementations can therefore be used for online estimation of the transition probabilities in time-critical applications in practice, which is not possible for a CPU implementation that can take more than 1 min for the same task.

## Acknowledgment

The authors would like to thank J. Portes for setting up machines and NAG for providing the `CUDA` routines for the MRG32k3a generator.

## References

[1] V. Bally and G. Pagès. A quantization algorithm for solving multi-dimensional discrete-time optimal stopping problems. *Bernoulli*, 9(6):1003–1049, 2003.

[2] V. Bally, G. Pagès and J. Printems. A quantization tree method for pricing and hedging multidimensional American options. *Math. Finance*, 15(1):119–168, 2005.

[3] O. Bardou, S. Bouthemy and G Pagès. Optimal Quantization for the Pricing of Swing Options. *Applied Mathematical Finance*, 16(2):183–217, 2009.

[4] O. Bardou, S. Bouthemy and G Pagès. When are Swing options bang-bang? *International Journal of Theoretical and Applied Finance (IJTAF)*, 13(06):867–899, 2010.

[5] A. L. Bronstein, G. Pagès and B. Wilbertz. How to speed up the quantization tree algorithm with an application to swing options. *Quantitative Finance*, 10(9):995 – 1007, November 2010.

[6] J. F. Carriere. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: Mathematics and Economics*, 19(1):19–30, 1996.

[7] J. H. Freidman, J. L. Bentley and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[8] S. Graf and H. Luschgy. *Foundations of Quantization for Probability Distributions*. Lecture Notes in Mathematics $n^0$1730. Springer, Berlin, 2000.

[9] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *OPERATIONS RESEARCH*, 47(1):159–164, 1999.

[10] P. L'Ecuyer, R. Simard, E. J. Chen and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *OPERATIONS RESEARCH*, 50(6):1073–1075, 2002.

[11] F. A. Longstaff and E. S. Schwartz. Valuing american options by simulation: A simple least-squares approach. *The Review of Financial Studies*, 14(1):113–147, 2001.

[12] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[13] G. Pagès and J. Printems. `www.quantize.maths-fi.com`. website devoted to quantization, 2005. maths-fi.com.

[14] J. L. Snell. Applications of martingale system theorems. *Trans. Amer. Math. Soc.*, 73:293–312, 1952.

[15] J. N. Tsitsiklis and B. Van Roy. Regression methods for pricing complex american-style options. *IEEE Transactions on Neural Networks*, 12:694–703, 2000.